ImitGraphs: towards faster usability tests of graphical model manipulation techniques

Ghazi, Parisa ; Glinz, Martin

Abstract: Due to the increasing use of general purpose graphical models (e.g., UML diagrams) and domain specific graphical models in different stages of software development, software engineers who work with these models spend more time interacting with modeling tools. Thus, the usability of interaction techniques of modeling tools affects the overall productivity of software development. Tool developers and user interface designers rely on the feedback from usability tests to optimize the user interface of tools that provide a graphical editor. Developing a working prototype to test new techniques is costly due to the complexity and variety of graphical models. This results in either tests at the late stages of development when changes are more expensive, or tests with prototypes that only support a subset of the intended graphical models, thus not comprehensive. In order to simplify conducting usability tests, instead of using the intended graphical models in the tests, we propose to use simpler models that require similar interactions when being manipulated. For this purpose, we introduced ImitGraphs, an extended graph with additional properties so that it can be specialized in interacting similarly to an intended graphical model. Then, we designed a method to instruct test participants to create ImitGraphs. Specialized graphs enable us to develop prototypes for usability tests faster and consequently cheaper resulting in more usability tests at early stages of tool development and on a wider range of intended models.

DOI: https://doi.org/10.1109/MiSE.2017.2

ImitGraphs: Towards Faster Usability Tests of Graphical Model Manipulation Techniques

Parisa Ghazi, Martin Glinz
Department of Informatics
University of Zurich
Zurich, Switzerland
{ghazi, glinz}@ifi.uzh.ch

***Abstract*—Due to the increasing use of both general-purpose and domain-specific graphical models (e.g., UML diagrams or graphic DSLs) in different stages of software development, software engineers who work with these models spend more time interacting with modeling tools. Thus, the usability of the interaction techniques employed by modeling tools affects the overall productivity of software development. Tool developers and user interface designers rely on the feedback from usability tests to optimize the user interface of tools that provide a graphical editor. Developing a working prototype to test new techniques is costly due to the complexity and variety of graphical models. This results in either tests at the late stages of development when changes are more expensive, or tests with prototypes that only support a subset of the intended graphical models. In order to simplify conducting usability tests, instead of using the intended graphical models in the tests, we propose to use simpler models that require similar interactions when being manipulated. For this purpose, we introduce graphs with additional properties, which we call ImitGraphs. ImitGraphs can be parametrized such that their interaction behavior is similar to that of an intended graphical model. Further, we introduce a method to instruct test participants to create ImitGraphs and manipulate them. ImitGraphs enable tool builders to develop prototypes for usability tests faster and consequently cheaper, thus resulting in more usability tests at early stages of tool development and on a wider range of intended models.**

***Keywords*-graphical model; software engineering; modeling tool; usability test; graph**

I. Introduction

Graphical models are used in various stages of software development, such as requirements engineering, software design, implementation, test, and maintenance. Therefore, the need for an effective and efficient user interface for manipulating these models is eminent. Designing a new user interface technique with a high usability is an iterative process [1], which requires multiple cycles of testing and improving. In such tests, participants are asked to perform predefined tasks using traditional and new techniques, giving the developers the chance to measure the improvement their techniques bring. Ideally, usability tests should cover all of the prospective graphical models [2]. However, since conducting such tests is expensive, designers either test a subset of their intended graphical models or test at a late stage of tool development.

The high cost of usability tests is due to the complexity and variety of graphical models. Thus, implementing a working prototype that handles the intended complex models becomes a demanding task. Eventually, user interface designers end up with complex code for their prototypes, which is difficult to change after receiving feedback from the tests. The possibility of widespread changes being required makes tool developers unwilling to invest in a comprehensive prototype. Instead, they prefer to test their interaction techniques on a specific type of graphical model [3]–[6].

To circumvent the complexity of graphical models, their equivalent simple graphs are already used for layout optimization purposes [7], but not for user interface usability tests. Conducting user interface usability tests would be much easier, if they could be conducted on graphs instead of the graphical models. In that case, tool developers could implement new techniques very quickly for simple graphs, resulting in lower cost or more tests, and would allow their techniques to mature. However, this is not possible due to the high degree of simplicity in graphs compared with graphical models. The simple nature of graphs results in needing simpler manipulation techniques. Therefore, an optimized user interface for graphs is not necessarily appropriate for graphical models and similarly, the feedback from usability tests on simple graphs is not completely valid for complex graphical models. Inspired by this idea, we asked this question: *"Can we define a special type of graph that is simple but has enough complexity to model graphical models?"* If such a type of graph can be defined, usability testers can use it in user interface usability tests resulting in a faster implementation of a working prototype and a higher number of testing iterations. Finally, after achieving an effective and efficient user interface, they can implement it for the original graphical models.

In this research, our goal was to study graphical models used in software development and find a way to define a special type of graph that can be used in usability tests of manipulation techniques instead of graphical models. To achieve this goal, we studied the process of manipulating diagrams and the interaction steps that a modeler takes. We found why simple graphs behave differently compared to their original complex graphical models. Based on these findings we defined an extended type of graph that can be specialized to imitate the behavior of graphical models in tests and named it ImitGraph.

Our contributions are: (i) the definition of ImitGraphs, (ii) a method to specialize them, (iii) a set of commands to instruct participants of usability tests to draw ImitGraphs similar to

the way they draw graphical models.

II. Related Work

Studies on human-computer interactions have provided principles of designing user interfaces with high usability [8]. However, even after following their guidelines the usability of the software product need to be evaluated and improved [1]. Usability testing is a fundamental way of usability evaluation [2]. When usability evaluation is carried out at a late stage, changes to the interface can be costly and difficult to implement [9]. Therefore, various methods for acquiring early feedback are proposed, e.g., rapid prototyping techniques [10]. Although the goal of our study, which is making early usability tests possible, is similar to those methods, we exploit the properties of the graphical models that our targeted modeling tools should handle.

Most of the research on enhancing the user interface of modeling tools focused on the visualization, navigation and rarely on manipulation techniques, e.g., onion graphs for visualizing UML class diagrams [11], semantic zooming for navigating UML diagrams [12] and off-screen visualization technique for UML class diagrams [3]. Despite these efforts, working with user interfaces of modeling tools is still considered as arduous [13].

Another group of studies enhanced the visualization by optimizing the layout of graphical models using graph layout algorithms [7]. In these works, the information about the elements of the graphical models is transferred to graph layout libraries along with settings and configurations, and these libraries return an optimized layout. Graphviz [14] is a popular library in this field. In addition, there are other works that created dedicated algorithms for automatic layout of specific graphical models such as class diagrams [4], use case diagrams [5] and data flow diagrams [6].

Most authors of related works have evaluated their proposed approaches on one type of graphical model only due to the expensive evaluation experiments. In this study, our goal is to benefit from the simplicity of graphs not only for layout optimization but to make usability tests faster on a wide range of graphical models without the need for an ad-hoc transformation implementation.

III. Graphical Models and Graphs

Evaluating new interaction techniques requires testing experiments which are costly. One of the main reasons of their high cost is the complexity of the graphical models for which a working prototype should be created. Therefore, we set our goal to find a suitable substitute to represent graphical models in such experiments. Specifically, our goal was to find a model with the following properties: (i) imitate the behavior of graphical models when being manipulated, (ii) be simple enough to allow quick implementation of new interaction techniques, (iii) has the potential to represent a large group of graphical models, and (iv) be easy to learn for the participants.

Since a simple graph already meets the requirements (ii)-(iv), it is a good starting point. We will discuss the remaining requirement, i.e. its behavior, in the rest of this section.

A. Diagram Manipulation in an Experiment

In order to explain a sample behavior of graphical models during manipulation, we made the following hypothetical scenario, in which the participant of a test is given the description of a process and is asked to draw a UML activity diagram. The following process describes how the system issues an invoice in an online shop:

The system first receives an order from a user, then, it issues the invoice. Before issuing the invoice, if the user is a member, the system applies a discount. In parallel to checking the user's membership and applying the discount, the system estimates the delivery date to be included in the invoice.

In our hypothetical scenario, the participant reads the process description and models the acquired information in an activity diagram as he goes on. After reading each part of the description and analyzing it, he applies the required changes in his mind. At certain points, he decides to transfer the changes from his mind to the actual diagram. This scenario is presented in Figure 1. The first column shows the description and is separated at the points where the text is meaningful. The second column is separated at points where the participant decides to transfer the changes from his mind to the actual diagram. If the tool does not offer any feature to apply the changes at once, he breaks down what he wants to do into smaller steps that his tool supports. Each cell of the third column shows the steps that the tool allows to be done at once. The fourth column shows the drawn diagram.

The part of this process that takes place in the mind of the test participant does not depend on the tool, but when transferring from the mind to the actual diagram, the breaking down depends on the features of the tool. We call the operations that are independent of the tool *Model-space operation*, and the operations that depend on the tool *Tool-space operation*.

Each model-space operation maps to one or more tool-space operations. Depending on the tool that is being used and the skill of the user, the tool-space operations fulfilling one model-space operation may differ. For example, Figure 2 shows two possible ways of inserting a decision element between two already existing activities in a UML activity diagram. In one way, firstly, the connection between two activities is removed, secondly, a new decision node is created, and finally, the activities and the decision element are connected together. In the other way, the decision element is created first, and then, it is dragged over the connection between the two activities. The tool breaks the connection automatically and creates two new connections accordingly. In this example (Figure 2), when only considering diagram "a" that should be transformed into diagram "b", it is a model-space operation and does not depend on the tool being used. However, the intermediate steps are tool-space operations. The modeler can choose one of these two ways.

We define the behavior of a graphical model as the mapping between model-space operations and the sets of tool-space operations that fulfill those model-space operations. In other words, we consider the behavior of two types of models to

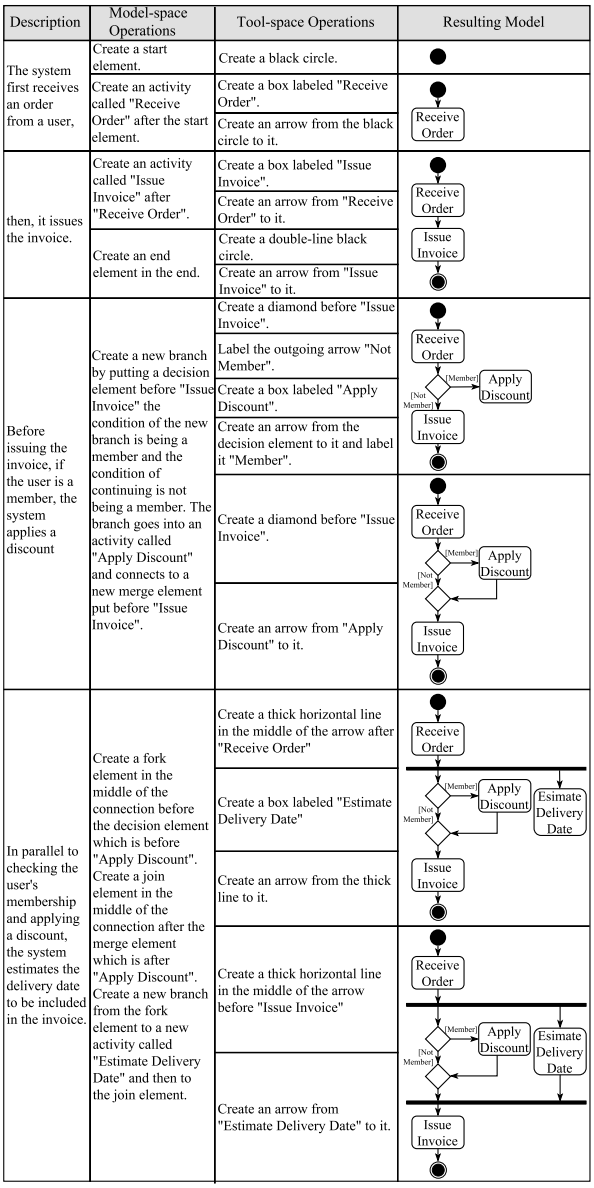

Fig. 1. A sample scenario of drawing an activity diagram based on a natural language description of the underlying process

be similar when two equivalent model-space operations are done by equivalent sets of tool-space operations. We continue by discussing two examples of how simple graphs behave differently from software engineering graphical models.

B. The Behavior of Simple Graphs

When a user interface designer wants to add a new feature such as the second insertion technique described in Figure 2 to his tool, he needs to be sure about the effectiveness of that feature before implementation. Therefore, he conducts tests with a prototype of the new technique. Figure 3 shows a part of a possible testing experiment when the tool designer uses a simpler equivalent graph instead of the activity diagram in order to implement the prototype quickly. The participant is given diagram "a" and is asked to insert another node between

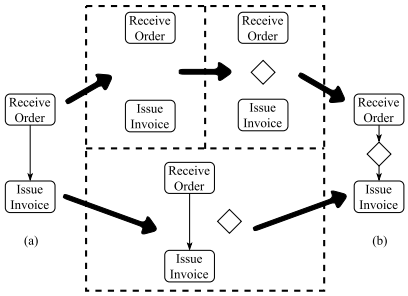

Fig. 2. Inserting a decision node between two activities in two ways: (i) removing the connection, creating the decision node and connecting them together, and (ii) creating the decision node, dragging it over the connection and the tool automatically breaks the connection into two connections.

the existing nodes so that it transforms into diagram "b". While the testers expect the participants to do this task in one of the ways shown in Figure 2, they may do it as shown in Figure 3, which is not possible in the activity diagram example of Figure 2. The reason for this difference is that duplicating the node B, connecting the new node to the existing node B and renaming the existing node B to C is easier. Since the sets of tool-space operations that fulfill the model-space operations in the activity diagram and the simple graph are different, the behavior of these two models is not similar. Therefore, the result of this experiment is not valid for activity diagrams.

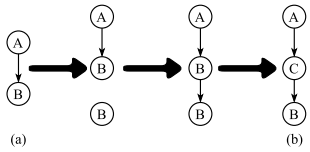

Fig. 3. Inserting a node between two existing nodes in a simple graph can be done differently from the similar example of Figure 2

Figure 4 shows another example of simple graphs' behavior. In this example, the size of an element affects the behavior of the graphical model and consequently affects the user interaction. The task of this experiment is to add another activity after activity A3 so that diagram "a" transforms into diagram "c". Before adding the new activity element, the participants need to provide some space by moving the other elements. However, if the experiment is done on the equivalent graph "b" instead of an activity diagram, adding the node E after the node B can be done without moving other nodes resulting in "d". Due to the difference in behaviors of simple graphs and activity diagrams, conclusions from this experiment are not equally valid for activity diagrams.

By these two examples, we showed how simple graphs behave differently than the graphical models. Therefore, despite meeting other requirements, they cannot represent graphical models in manipulation usability tests. In the next section, we describe how we compensate for this shortcoming.

IV. Our Approach

Our proposal includes an extended definition of graph called *ImitGraph*, a way to specialize ImitGraphs for different

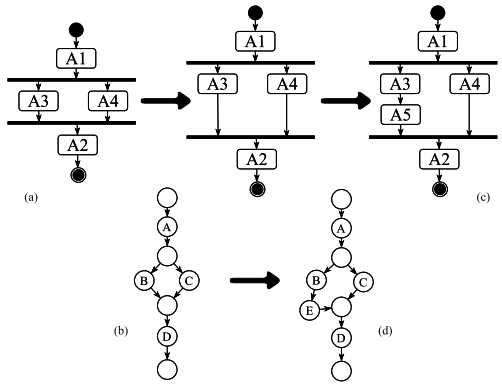

Fig. 4. Similar tasks on an activity diagram and its equivalent graph may result in different layouts

purposes and a set of commands to define model-space operations. If a usability tester wants to use ImitGraphs defined in Section IV-A, he should first define his desired types of nodes and connections using the specialization method of Section IV-B, and then, design tasks for participants using the ImitGraph commands introduced in Section IV-C.

A. Definition of ImitGraphs

Inspired by the simplicity of the graphs, we extended the definition of graphs by adding more properties to the nodes and connections. The additional properties allow the usability testers to specialize ImitGraphs depending on the purpose of their tests and the graphical models involved.

ImitGraphs are composed of nodes, connections and joints. Figure 5 shows examples of each element.

1) Node: A node is a circular element that can be assigned different sizes, colors, and it can hold a label.

2) Connection: A connection is a line with a rectangle in the middle. It connects two joints. The rectangle can be assigned different colors and a label.

3) Joint: A joint is a circle at the end of a connection. Joints attach connections to nodes. They can be assigned different colors and labels.

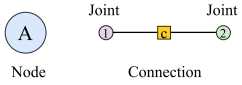

Fig. 5. A node and a connection of a specialized graph

Since joints are used in defining connections and connections are used in defining the nodes, we first describe the properties of joints, then connections and finally the nodes.

Joints have the following properties: (i) Color: the color of a joint enables the user to distinguish different types of joints. (ii) Label: if a graphical model requires joints to have textual parts (e.g., the cardinalities in a class diagram), their equivalent ImitGraph joints should hold labels instead.

Connections have the following properties: (i) Color: the color of the rectangle enables the user to distinguish different types of connections. (ii) Label: if a graphical model requires the connections to have a textual part (e.g., the conditions in a flow chart diagram), their equivalent ImitGraph connections should hold a label. (iii) First joint: indicates the type of the joint at one end of a connection. (iv) Second joint: indicates the joint type for the other end of the connection. (v) Orientation: shows if a connection can be oriented in any direction or it is restricted to certain orientations (e.g., horizontal).

Nodes have the following properties: (i) Color: each node type has a different color so that the users can recognize their types. (ii) Size: if the size of the nodes matter in an experiment they can be defined differently, otherwise, similar node sizes make experimenting simpler. (iii) Label: if the original graphical model's counterpart element holds a text, the ImitGraph node should hold a label. (iv) Connection type: each node is restricted to be connected with other nodes with certain types of connections. (v) Joint type: for each connection type, it indicates which joint of the connection should be connected to the node. (vi) Connection point: indicates if the connection can be connected to the node at any point on the perimeter or it is restricted to certain points (e.g., decision nodes in activity diagrams).

In graphical models, the group of elements that have a text can be referenced directly. The other elements are referenced relative to the nodes of the first group. This behavior is simulated by the ability of the nodes, joints, and connections to accept a label or not.

B. Specialization of ImitGraphs

Before usability testers use ImitGraphs in their experiments, different types of nodes, connections and joints should be defined by specifying the properties of each type. The properties are specified based on the elements and connections of the graphical models on which interaction techniques are going to be tested. Types should be defined adequately so that the behavior of the intended graphical models can be simulated. We called this phase *Specialization*.

Figure 6 shows two examples of ImitGraphs and their original graphical model: "a" is a part of an entity-relationship diagram (ERD) and "b" is a part of an activity diagram. Usability testers specify different colors of the elements and whether they accept labels or not. The colors of the nodes in the ERD diagram's equivalent ImitGraph are the same since the nodes are of the same type. In contrast, the colors of the nodes in the activity diagram's equivalent ImitGraph are different. The colors of the joints in both ImitGraphs are different due to the different types of endpoints in connections of the original models (e.g., flat, arrow and trifurcation). The nodes, joints, and connections have labels if their counterpart elements have texts.

The definitions of nodes and connections are reusable. Once the nodes and connections of a certain type of graphical model are defined, they can be used in other experiments that include the same type of graphical model.

C. Instructing Commands to Draw ImitGraphs

To make ImitGraphs applicable in usability tests we defined a set of commands for specifying the model-space operations.

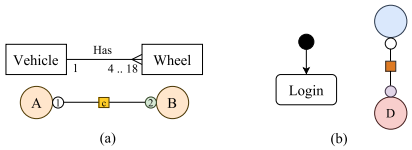

Fig. 6. Parts of "a" an entity-relationship diagram and "b" an activity diagram, and their equivalent specialized graphs

The usability testers use ImitGraph commands to instruct the participants to draw a graph. Although the graph that should be drawn in a testing experiment is equivalent to a software engineering graphical model, the participants are not aware of that. Since ImitGraphs do not have the semantics of their originals, the instructing methods of the original graphical models, e.g., the natural language description used in Figure 1, are not applicable. Therefore, we defined a set of ImitGraph commands to instruct the participants of tests. These commands do not suggest (i) a specific layout, (ii) specific tool-space operations, and (iii) a specific order of operations. The end user should have a similar freedom of choosing any tool-space operation, order and layout for drawing as he has when drawing based on a natural language description. They even should be able to make similar errors.

Before explaining the commands, we define two terms. *Current location* can be a node or a connection. It is the last node created, the node referenced by the last "Find Node" command, or the connection referenced by the last "Find Connection" command. *Referenceable location* is a type of node that can be located unambiguously by the participants. It can be a node with a label, current location, or a memorized location specified by "Remember as" command.

As the examples in Figure 7 show, the ImitGraph commands are partially textual and partially visual. Subsequently, we describe the commands in more detail:

1) Create: As shown in Figure 7a, this command is followed by a node symbol with a color and a label if required. It instructs the participants to create a new node with the specified properties. It does not suggest any tool-space operation (e.g., it can be created by duplicating an existing node) or a position.

2) Branch: As shown in Figure 7b, this command is followed by a referenceable node. Then it continues with a sequence of connections and nodes. This command instructs the participants to add a branch made of nodes and connections to the diagram, which starts from a known node and ends in a known node or a new node. The participant is free to start from any of the connections or nodes of the sequence.

3) Find Node: Except for the nodes that are directly referenceable, other nodes should be referenced relatively. As shown in Figure 7c, this command is followed by a node. The node can be referenceable or not. If the specified node is referenceable, this command instructs the participants to consider it as the current location. Otherwise, the participant should find a node of the specified type that is connected to the current location and consider it as the current location.

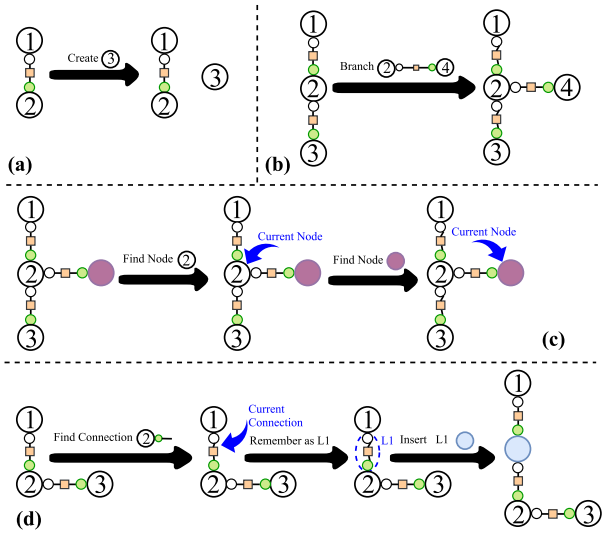

Fig. 7. Examples of specialized graph commands that instruct participants how to manipulate diagrams: (a) Create a new node, (b) Branching from an existing node, (c) finding a node that is not referenceable, (d) finding a connection, assigning it a label and inserting a new node in the middle of it.

4) Find Connection: As shown in Figure 7d, this command is followed by a referenceable node. Then the node is followed by a joint. This command instructs the participants to find the specified node, and then, find the connection that is connected with the specified joint to it. The found connection should be considered as the current location.

5) Remember as: As shown in Figure 7d, this command instructs the participants to assign the specified name to the current location in their mind. If the current location was not referenceable before, it can be referenced by this name afterward.

6) Insert: When this command is used, the current location should be a connection. As shown in Figure 7d, this command is followed by a new node and after that a sequence of connections, and nodes. It instructs the participants to add a new branch to the diagram. The beginning of the branch is a new node that should be placed between the nodes connected by the connection known as the current location at the moment. The end of the branch can be a new node or a referenceable existing node. This command does not suggest any order, layout or tool-space operation for creating the nodes and connections of the new branch.

7) Recent: This command is followed by a node type that is not referenceable. It is used to reference a node that is created by the previous command and is basically not referenceable.

8) Other commands: Remove instructs the participants to delete the current location. If the current location is pointing to a node, its connections should be deleted too. *Connect* followed by a referenceable node, a connection and another referenceable node, instructs the participant to create a connection between two existing nodes. *Replace* followed by a node or a connection, instructs the participants to replace the current location with the specified node or connection.

V. A Sample Usage Scenario

In this section, we demonstrate how usability testers can use ImitGraphs in usability testing experiments. For this purpose, we made a hypothetical experiment, in which the usability testers intend to test the usability of their tool's interaction techniques when manipulating activity diagrams. For this purpose, they ask the participants to draw an ImitGraph. Before the tests, the testers specialize the ImitGraph to imitate the behavior of activity diagrams by defining equivalent joints, connections, and nodes. Then, they use ImitGraph commands to create a task based on the process of Figure 1.

The participants are not aware of the relationship between the graph that they draw and the activity diagram. They first study the definition of the nodes, connections and joints. Then, they read the commands one by one like a natural language description, and at certain points decide to perform the commands that they have read. Since the commands only instruct the participants to imagine an addition or change, the decision of how to perform them is made based on the available tool features. The first column of Figure 8 shows the task made of commands. The second column shows how participants can split the task into model-space operations. The third column shows tool space operations that participants perform to fulfill each model-space operation. In this example, we assumed that the tool under test is a simple drawing tool that can create nodes, connect them and insert a node in the middle of a connection. Different participants can split the task in different ways and perform the model-space operations with different tool-space operations. Figure 8 only shows one possible way.

The specialized ImitGraph defined by the usability testers based on our activity diagram example (Figure 1) includes two types of joints which are presented in Table I, two types of connections which are presented in Table II, and six types of nodes which are presented in Table III.

In Table I, the white joint type (J1) represents the simple flat end of the connections in activity diagrams. The green joint type (J2) represents the arrow end of the connections. None of the joints can have a label in this example.

In Table II, the beige connection type (C1) represents the connections of activity diagrams which have a flat end (J1) and an arrow end (J2). They can be oriented in any angle and can hold a label when they are connected to the equivalents of decision nodes. The red connection type (C2) is used to represent fork/join elements of activity diagrams. They are flat at both ends (J1) and only can be oriented horizontally.

TABLE I
Joint Types

Type	Symbol	Label
J1		no
J2		no

TABLE II
Connection Types

Type	Symbol	Label	First joint	Second joint	Orientation
C1		optional	J1	J2	any
C2		no	J1	J1	horizontal

In Table III, the gray node type (N1) represents the start element of activity diagrams which does not accept a label.

TABLE III
Node Types

Type	Symbol	Size (pixel)	Label	Connection type	Joint type	Min	Max	Connection point
N1		30	no	C1	J1	1	1	any
N2		30	no	C1	J2	1	1	any
N3	n	30	yes	C1	J2	1	1	any
				C1	J1	1	1	any
N4		30	no	C1	J2	0	1	top
				C1	J1	0	1	bottom
				C2	-	1	2	left, right
N5		30	no	C1	J2	1	1	top, left, right, bottom
				C1	J1	2	3	top, left, right, bottom
N6		30	no	C1	J2	2	3	top, left, right, bottom
				C1	J1	1	1	top, left, right, bottom

It can have only one simple arrow connection (C1), which connects to the node by its flat end (J1) and at any point of the node's perimeter. The purple node type (N2) represents the end element of activity diagrams. It is similar to N1 but connects to the arrow from the arrow end (J2). The white node type (N3) represents the activity elements of activity diagrams. They hold labels and have one incoming connection and one outgoing connection of type C1. Incoming connections connect with a J2 joint and outgoing connections connect with a J1 joint. The connections can connect at any point to this type of nodes. The blue node type (N4) are used in creating equivalents of fork/join elements of activity diagrams. Multiple nodes of this type represent one fork/join element. For this purpose, they are connected together with C2 connections that can only be oriented horizontally and are connected only from left and right. The number of N4 nodes that represent a fork/join element depends on the number of incoming and outgoing connections of the fork/join element. The incoming connections of type C1 connect with a J2 joint to the top. The outgoing connections of type C1 connect with a J1 joint to the bottom. The orange (N5) and the yellow (N6) node types represent decision and merge elements of activity diagrams. Therefore, they are restricted to be connected at the top, left, right and bottom only.

This example shows that specialized ImitGraphs can be drawn in the same way as their original graphical model. In addition, ImitGraph commands can instruct the participants to draw a graph without suggesting any layout, operation or order. The participants are responsible for choosing appropriate tool-space operations. The definition of the joints, connections, and nodes make the participants draw the diagrams with a layout similar to the original graphical model.

VI. Conclusions and Future Work

User interface researchers and tool developers who work on improving the usability of software modeling tools need to conduct tests to evaluate the effectiveness of their new ideas and gather feedback to improve them. In this paper, we proposed ImitGraphs, an extended version of graphs that can

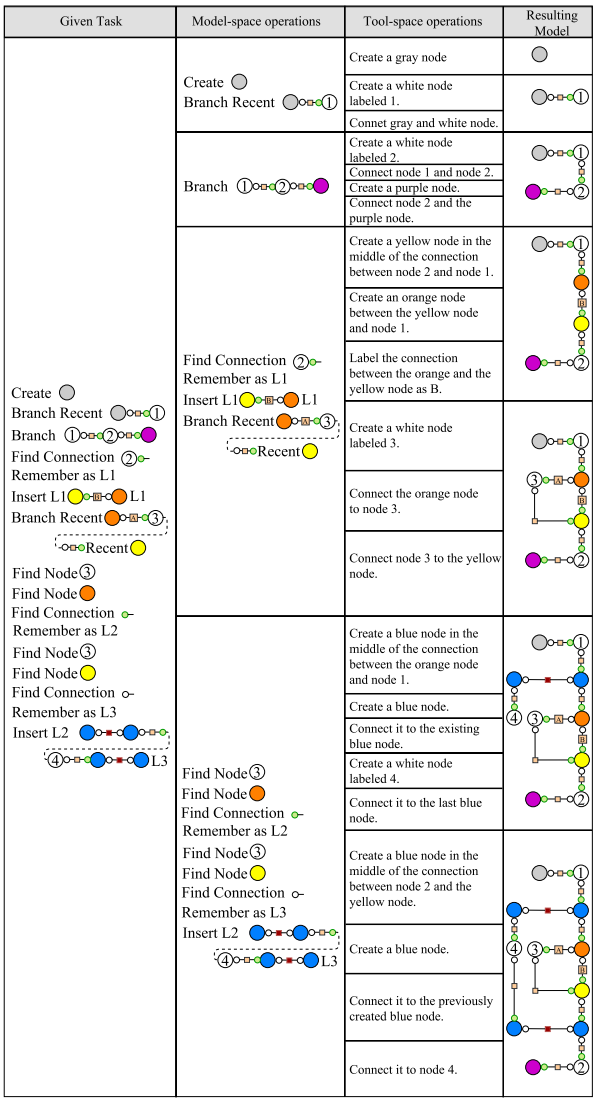

Fig. 8. Equivalent scenario of Figure 1 in ImitGraph notation. The first column contains the task given to the participants. The second column shows the detected model-space opreations by the participant. The third column shows the corresponding tool-space operations. The last column shows the resulting diagram at each step.

be used in such tests instead of real graphical models. The benefits of using this approach are: (i) fast development of a working prototype to evaluate new ideas resulting in a lower cost and earlier feedback, (ii) evaluating the effectiveness of the new ideas on a wider range of graphical models, and (iii) possibility to recruit participants with no prior knowledge about the intended graphical models for the experiments. In exchange for these benefits, tool developers (a) have to teach ImitGraphs definition and its commands to the participants before their first experiment, and (b) must develop a working prototype for ImitGraphs which is not a part of the final product. The former can be compensated by enabling testers to recruit participants more easily since more people fit. The latter can be justified by the low cost of developing a tool for ImitGraphs due to their simple appearance and the fact that the developed prototypes and the definitions of the ImitGraphs will not be thrown away. They will be used in further optimizations and tests of future improvements of the user interface. Even other user interface designers and researchers can benefit from the already developed prototypes and definitions to rapidly test their ideas at a very early stage.

Since ImitGraphs imitate the interaction behavior and layout properties of graphical models, the gained experience and feedback can be transferred to the original models. Therefore, ImitGraphs can be used to test a wide range of user interface interaction features such as drawing commands, automatic alignment of elements, placement of connections, readjustment of the layout when changes occur or when more space is required and providing frequently done operations as a single command. The presentation of the tools such as placement of the buttons and activators, the structure of the menus, the expressiveness of the icons, the way of showing hints and extra information can be tested, in addition to their features. Furthermore, complex features such as graphical search and filter can be optimized by conducting tests using ImitGraphs.

This study will continue with specializing ImitGraphs for some of the most frequently used graphical models in software engineering, developing tool-support for manipulating ImitGraphs, and finally, use them in real experiments to evaluate ImitGraphs.

References

[1] D. J. Mayhew, "The usability engineering lifecycle," in *CHI'99 Extended Abstracts on Human Factors in Computing Systems*. ACM, 1999, pp. 147–148.

[2] J. R. Lewis, "Usability testing," *Handbook of Human Factors and Ergonomics*, vol. 12, p. e30, 2006.

[3] M. Frisch and R. Dachselt, "Off-screen visualization techniques for class diagrams," in *Proceedings of the 5th International Symposium on Software Visualization*. ACM, 2010, pp. 163–172.

[4] M. Eiglsperger, "Automatic layout of UML class diagrams: a topology-shape-metrics approach," Ph.D. dissertation, Universität Tübingen, 2003.

[5] H. Eichelberger, "Automatic layout of UML use case diagrams," in *Proceedings of the 4th International Symposium on Software Visualization*. ACM, 2008, pp. 105–114.

[6] C. D. Schulze, "Optimizing automatic layout for data flow diagrams," Ph.D. dissertation, Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2011.

[7] M. Spönemann, *Graph Layout Support for Model-Driven Engineering*. BoD–Books on Demand, 2015.

[8] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.

[9] A. Holzinger, "Usability engineering methods for software developers," *Communications of the ACM*, vol. 48, no. 1, pp. 71–74, 2005.

[10] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[11] H. Kagdi and J. I. Maletic, "Onion graphs for focus+context views of UML class diagrams," in *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2007, pp. 80–87.

[12] M. Frisch, R. Dachselt, and T. Brückmann, "Towards seamless semantic zooming techniques for UML diagrams," in *Proceedings of the 4th International Symposium on Software Visualization*. ACM, 2008, pp. 207–208.

[13] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill *et al.*, "The relevance of model-driven engineering thirty years from now," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 183–200.

[14] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and Dynagraph – static and dynamic graph drawing tools," in *Graph Drawing Software*. Springer, 2004, pp. 127–148.